REMARKS

Claims 1-3, 6-8 and 11-24 are pending in the present application. By this reply, claims 4, 5, 9 and 10 have been cancelled and claims 17-24 have been added. Claims 1 and 17 are independent claims.

The specification and claims have been amended to improve their form and to clarify the invention. The drawings have also been amended as indicated in red ink to clarify the invention. A Drawing Change Approval Request (DCAR) is concurrently filed herewith. The Examiner's approval of the drawing corrections is respectfully requested.

35 U.S.C. § 103 REJECTION

Claims 1-5, 8-13 and 15-16 have been rejected under 35 U.S.C. § 103(a) as being unpatentable over Applicant's disclosed background art[1] and in view of *Farrell et al.* (U.S. Patent No. 5,247,675). This rejection is respectfully traversed.

The Examiner correctly acknowledges that Applicant's disclosed background art does not teach inserting the tasks into a waiting list in priority order, such that the task having the highest priority in the waiting list obtains the event, is woken up and resumes execution. To overcome these deficiencies, the Examiner further relies of *Farrell et al.*

[1] It is not certain whether Applicant's disclosed background art qualifies as prior art under 35 USC 102. But for the sake of the argument, Applicant will address the rejection.

Farrell et al. is directed to storing the highest priority thread from each dispatch class onto a run list, where the run list is executed. The Examiner equates this run list to Applicant's waiting-list of the event. However, *Farrell et al.*'s run list queues therein only those threads with the highest priority from each dispatch class. In contrast, Applicant's waiting-list stores therein all the tasks (and not just the highest priority tasks) that did not receive the event. Claims states provided Thus, *Farrell et al.*'s run list and its operation are different from Applicant's waiting-list and its operation.

If Applicant's disclosed background art were modified to include *Farrell et al.*'s run list and its operation as suggested by the Examiner, then the combination of references does not teach or suggest, *inter alia*:

> calling, by each of a plurality of tasks, a kernel system function of receiving an event with respect to one event under the multi-tasking environment; and
>
> blocking said each of the tasks and inserting said each of the tasks into a waiting-list of the event in priority order when no event is provided to the tasks

as recited in independent claim 1.

Accordingly, claim 1 and its dependent claims (due to their dependency) are patentable over the applied references, and reconsideration and withdrawal of the rejection based on these reasons is respectfully requested.

Claims 6, 7 and 14 have been rejected under 35 U.S.C. § 103(a) as being unpatentable over Applicant's disclosed background art[2] and in view of *Farrell et al.* (U.S. Patent No. 5,247,675), and further in view of *Saulpaugh* (U.S. Pat. No. 5,734,903). This rejection is respectfully traversed.

As discussed above, the combination of Applicant's disclosed background art and *Farrell et al.* fails to teach or suggest, *inter alia*:

> calling, by each of a plurality of tasks, a kernel system function of receiving an event with respect to one event under the multi-tasking environment; and
>
> blocking said each of the tasks and inserting said each of the tasks into a waiting-list of the event in priority order when no event is provided to the tasks

as recited in independent claim 1 from which claims 6, 7 and 14 depend. *Saulpaugh* does not overcome these deficiencies since *Saulpaugh* is merely directed to generating a message ID and error code. Thus, even if the references are combinable, assuming *arguendo*, the combination does not teach or suggest at least the above-identified features of claim 1.

Accordingly, claim 1 and its dependent claims (due to their dependency) are patentable over the applied references, and reconsideration and withdrawal of the rejection based on these reasons is respectfully requested.

---

[2] It is not certain whether Applicant's disclosed background art qualifies as prior art under 35 USC 102. But for the sake of the argument, Applicant will address the rejection.

CONCLUSION

For the foregoing reasons and in view of the above clarifying amendments, Applicant respectfully requests the Examiner to reconsider and withdraw all of the objections and rejections of record, and earnestly solicits an early issuance of a Notice of Allowance.

Should there be any outstanding matters which need to be resolved in the present application, the Examiner is respectfully requested to contact Esther H. Chong (Registration No. 40,953) at the telephone number of the undersigned below, to conduct an interview in an effort to expedite prosecution in connection with the present application.

Pursuant to 37 C.F.R. §§1.17 and 1.136(a), Applicants respectfully petition for a one-month extension of time in which to file this reply. A check for the $110 fee is attached.

If necessary, the Commissioner is hereby authorized in this, concurrent, and further replies, to charge payment or credit any overpayment to Deposit Account No. 02-2448 for any additional fees required under 37 C.F.R. § 1.16 or under 37 C.F.R. § 1.17, particularly extension of time fees.

Respectfully submitted,

BIRCH, STEWART, KOLASH & BIRCH, LLP

By Scott L. Lowe #41,458
Joseph A. Kolasch, #22,463

P.O. Box 747
Falls Church, VA 22032-0747
(703) 205-8000

630-1060P
Attachment
JAK:EHC:lh

VERSION WITH MARKINGS TO SHOW CHANGES MADE

In the Abstract

The Abstract of the Disclosure has been amended as follows:

--In the method for implementing the event transfer system of a real time operating system kernel, the task with the highest priority first obtains the event under the multi-tasking environment which requires [a] real time characteristics. In the case of the multi-tasking environment in which the priority-based preemptive scheduling is adapted, if a plurality of tasks with respect to one event, call a kernel system function of receiving the event, the real time operating system kernel queues the tasks into the waiting-list of the event in the priority order. In this state, when the event is sent, the task having the highest priority in the waiting-list immediately obtains the event, is woken up and is resumed in its execution.--

In the Specification

The specification has been amended as follows:

On page 3, lines 7-14 have been amended as follows:

The event control block is a real structure of an event and is formed of a data structure managed by the kernel. Creating an event control block means that a task creates an event of its own. In the case that the second task 2 with the priority higher than that of the first task is created and started to execute[,

when] and the second task 2 calls the kernel system function of waiting for (receiving) an event, then since the first task transfers no event yet, the second task 2 is blocked to the wait state waiting for the event to be transferred (sent) and is queued into the waiting-list of the event control block 1.

On page 3, line 20 – page 4, line 8, have been amended as follows:

Thereafter, when the first task starts to send the event, at first the second task receives [to obtain] the event, is woken up and is resumed execution. Namely, the event is first transferred to the second task 2 first queued to the waiting-list based on the FIFO (First-In-First-Out), so that the second task is executed. Therefore, it causes the third and fourth tasks 3 and 4 having relatively higher priorities not to be first performed. Namely, the above-described method may be adapted to a known round robin scheduling method, but it may not satisfy the scheduling mechanism which supports [a] real time characteristics.

In order to overcome the above-described problem, the structure of the waiting-list should be formed in a doubly linked list, and when an event is sent, the task with the highest priority should be searched in the waiting-list and resumed to execute. However, in this case, since [it's] it is not until an event is sent that the searching of the waiting-list is performed, it may take more reaction-time. And the more number of the tasks waiting for the event, the more time it takes also.

On page 6, lines 8-13 have been amended as follows:

As shown in Figure 2, in this example, it is defined that the priority value of the first task is 40, the priority value of a second task 102 is 30, the [priorities] priority values of third and fourth tasks 103 and 104 are 20, respectively[,] (the lower priority values have the higher priority), and that the second through fourth tasks 102 through 104 are trying to receive the event which the first task transfers periodically through an event control block 101 of the first task.

On page 6, line 22 - page 7, line 19 have been amended as follows:

The task order queue in the waiting-list at the time when the second through fourth tasks 102 through 104 have been all blocked (E in [the] Figure 4), is not the sequence of calling the kernel system function as shown in Figure 1. Instead, the order [of] will be the third task 103, the fourth task 104 and the second task 102 which is the priority order of tasks as shown in figure 2. When a kernel system function call is performed for receiving an event by each task but [none of] no event is sent yet, each task is blocked and is set to the wait state. At this point, the priority of each task is checked and the task is inserted into the position of the priority order of the waiting-list so that the waiting-list becomes the priority order. Therefore, the waiting-list is always

maintained in the state of the priority order <u>such</u> that the task with the highest priority is placed at the head of the waiting-list.

When the first task calls a kernel system function of transferring (sending) the event from the point F of Figure 4, the priority check is not additionally needed with respect to the tasks of the waiting-list[, and the]<u>. The</u> task with the highest priority at the head of the waiting-list is picked up from the list, and the event value is transferred to the task, so that the <u>execution of the</u> task is resumed [execution]. It means that since the waiting-list is already aligned in the higher priority order, when transferring the event, the task with the highest priority of the waiting-list first obtains (receives) the event by merely waking up and resuming the head-positioned task.

As shown in Figure 4, the event transferred by the first task is alternatively obtained by the third task 103 and the fourth task 104 having the highest priority equally, and the second task 102 which has <u>the</u> relatively lower priority does not obtain the event.

On page 8, lines 11-20 have been amended as follows:

As a result of the check, if the event ID is invalid, it means an error situation [that] <u>exists where</u> the event-receiving attempt is performed with respect to the non-existing event. Therefore, the scheduling is enabled in Step S12, and the current task is returned from the kernel system function with an error code in Step S13.

As a result of the check, if the event ID is valid, next it is checked that whether the event value has been already transferred (sent) in Step S4. If the value exists, the event value is obtained from the buffer in the event control block in Step S5, [and] the process routine is performed by the kind of the event in Step S11, [and] the scheduling is enabled in Step S12, and the current task is returned from the kernel system function with the event value in Step S13.

On page 9, lines 9-13 have been amended as follows:

/* if the waiting-list is empty (any task that waits for a corresponding event does not exist), or if the priority of the current task is lower than the priority of the tail portion task of the waiting-list or is the same (the priority of the current task is lower than any other tasks already queued with the standby list), [it] the current task is directly inserted into the rear end (tail) of the waiting-list. */

On page 11, line 21 - page 12, line 1 have been amended as follows:

The head (leading) task of the waiting-list, namely, the task having the highest priority among the tasks of the wait state is woken up and is resumed in its execution when another task which is supported to send the event calls a kernel system function of sending the event or the time-out is elapsed to thereby cause the preemption[, and the]. The task routine is returned from the

kernel system function of receiving the event based on the steps S11 through S13.

On page 12, lines 4-19 have amended as follows:

When a certain task which is currently being executed calls the kernel system function of sending the event in Step ST1, the scheduling is temporarily disabled like the kernel system function of obtaining the event in Step ST2, and it is checked whether the argument ID of the event that the current task sends to is valid in Step ST3. As a result of the check, if the event ID is invalid, it means that an error situation [that] exists where the event-sending attempt is performed with respect to the non-existing event. Therefore, the scheduling is re-enabled in Step ST12, and the current task routine is returned from the kernel system function with the error code in Step ST13.

As a result of the check, if the event ID is valid, next it is checked whether the waiting task exists in the waiting-list of the event in Step ST4. if any task of the wait state does not exist with respect to the event, the event value is simply stored in the event buffer of the event control block in Step ST5, and the event kind-based process routine may be additionally performed in Step ST11[, and then]. Then the scheduling is re-enabled in Step ST12, and the current task routine is returned from the kernel system function in Step ST13.

On page 13, lines 12-15 have been amended as follows:

Thereafter, if the scheduling is re-enabled in Step ST12, the preemption can be occurred, so that the task (which was the head (leading) task in the waiting-list), which has been set ready to wake up in [the step] Step ST10 is resumed in its execution.

In the Claims

Claims 4, 5, 9, and 10 have been cancelled.

The claims have been amended as follows:

1. (Amended) In a method for implementing an event transfer system of a real time operating system kernel under a multi-tasking environment in which a priority-based preemptive scheduling is adapted, a method for implementing an event transfer system of a real time operating system kernel [which is characterized in that in the cast that], comprising:

calling, by each of a plurality of tasks, [call] a kernel system function of receiving an event with respect to one event under the multi-tasking environment[,]; and

blocking said each of the tasks [are blocked] and [previously inserted] inserting said each of the tasks into a waiting-list of the event in [a higher] priority order when no event is provided to the tasks, [and]

wherein in the case that the event transfer occurs, the task having the highest priority in the waiting-list obtains the event, is woken up and is resumed with execution.

2. (Amended) The method of claim 1, wherein said waiting-list of the event is managed based on the [higher] priority order so that the task having the highest priority is arranged at the most leading portion (head) of the waiting-list.

3. (Amended) The method of claim 1, [wherein] further comprising:

checking whether there is an event value already sent, when the kernel system function of receiving the event starts[, it is checked whether there is an event value already sent].

6. (Amended) The method of claim 1, [wherein when the kernel system function of receiving the event starts,] further comprising:

checking a validity of [an] the event ID [is further included] for thereby generating an error code in the case of validity when the kernel system function of receiving the event starts [, and]; and

returning the routine [is returned] from the kernel system function.

7. (Amended) The method of claim 13 [5], wherein when the current task is queued into the waiting-list, a time out option is additionally set if it exists.

8. (Amended) The method of claim 1, wherein when transferring the event, [it is checked] the method further comprises:

checking whether [the] any waiting task exists in the waiting-list of the event.

11. (Amended) The method of claim 15 [10], wherein said head task in the waiting-list which receives the event value is adjusted to a [the] ready state and is inserted into a [the] ready list, and an [the] additional process routine by the sort of the event is executed.

12. (Amended) The method of claim 3, wherein as a result of the check whether the event value exists, when the event value exists, the event value is obtained [from] an the event control block buffer, and the task routine is executed by the sort of the event.

14. (Amended) The method of claim 3, wherein when the kernel system function of receiving the event starts, the method further comprises:

[a step for] checking a validity of an [the] event ID [is further included] for thereby generating an error code in the case of invalidity[,]; and

returning the routine [is returned] from the kernel system function.

15. (Amended) The method of claim 8, wherein as a result of the check that whether the task exists in the waiting-list, when the waiting task does not exist, an event value is stored in [the] an event buffer of [the] an event control block.

New claims 17-24 have been added.